JavaBIP User Manual

- Introduction
  - BIP value statement
    - FSM
    - Coordination
- Behaviour Model
  - Basic behaviour for BIP components
  - Execution semantics for the basic behaviour
  - Guarded transitions
  - Data variables and actions
- Use case
  - Routes
  - BIP Control
  - Notifier
  - BIP Monitor
  - BIP model of the use case
- Software architecture
  - BIP API
    - BIPActor
    - BIPComponent API
    - BIP-engine API
    - BIPActorAware
  - BIP annotations needed for BIP Specification
    - ComponentType
    - Ports & Port
    - Transitions & Transition
    - Guard
    - ExecutableBehaviour
    - Data
      - Output data
      - Input Data
      - Guard Data
  - BIP Modules
- Glossary
  - BIP specific
    - Labelled Transition Systems (LTS)
    - Separation of concerns
    - BIP Component and Ports
    - Glue: Interactions and Priorities
    - Engine
    - Safety and Liveness Properties
    - BIP Monitor
    - BIP Module
  - OSGi specific
    - OSGi container
    - Bundle
    - OSGi registry

        - OSGi service
        - OSGi service property
    - Spring
        - Spring Application Context
        - Spring bean
        - Dependency Injection
    - Camel
        - Camel Context
        - Camel route
        - Endpoint
        - Exchange
- Addendum
    - BIP framework versus Akka
        - Coordination & BIP Glue
        - FSM
        - BIP protocol & BIP component
        - BIP Spec
    - BIP Presentations
        - Eclipse Conference 2013 - Technical Summary/Abstract
        - Internal presentation at CT

Introduction

The key difference between the usual BIP design flow and the CF/BIP integration project resides in the relation between the BIP model of the system and the executable code. The usual BIP design flow leads to the automatic generation of complete executable[1] code for the entire system. In the context of CF/BIP integration, the goal is to add BIP coordination to existing Java code running within the Connectivity Factory (CF). CF is an OSGi container with applications built using advanced frameworks, such as Spring (for dependency injection) and Camel (for data route management). On one hand, using these frameworks raises the abstraction level, thus simplifying the CF design process, but, on the other hand, the resulting executable code is not available for direct manipulation by the designers. More precisely, the route operation in Conectivity Factory is driven by one or several Camel contexts, implemented by the Camel library. Developers only provide route specifications composed of predefined building blocks. Furthermore, the best practice used in CF consists in using only well-defined approaches, namely Spring and OSGi registry, for object coupling. BIP Entities within CF must similarly be wired together with the help of Spring and OSGi registry. Thus, we can spell out the constraints that must be respected in the CF/BIP integration project:

1. OSGi service discovery functionality must be used to establish the links between BIP components and the engine;
2. Links between existing code and corresponding BIP components must be established through the Spring context;
3. BIP coordination of the CF routes can only be performed through the mechanisms and interfaces provided by the Camel context;
4. Glue definition mechanism must provide sufficient flexibility to enable defining coordination constraints at run time, when all the components are instantiated and dependencies established.

The first constraint is addressed by packaging the BIP Engine as an OSGi bundle and using the mechanisms provided by OSGi, namely Bundle Activator, to publish the service that later can be used by BIP components. To minimize the dependence of the BIP Engine on OSGi API, the use of OSGi mechanisms must be clearly separated from the functional code.

The second constraint is addressed trivially, provided that the code of the controlled entity is clearly separated from the code of the controlling BIP component. The reference to the controlled entity is accessible within Spring context and can be injected into another bean, in particular the BIP component specification bean.

The limitations imposed by the third constraint on how BIP components can interact with the controlled processes requires that BIP behaviour model be adapted correspondingly. Section 3 presents the proposed modification.

Finally, the fourth constraint is addressed by specifying glue as boolean constraints using Dynamic BIP macros approach. Precise way to specify glue constraints will be defined in the next version of this document.

BIP value statement

This section contains different discussion points to advertise BIP approach. It is meant to focus on technical aspects.

FSM

Interfaces are not enough. Interface is just a list of functions. It hides a state of the component that is behind the interface. Some of the functions may not be available and called at the wrong time violating the invariants of the implementation.

We expect to have explicitly stated FSM so that it can be analysed and used during runtime. The minimum plan is to have protection mechanism in place that will protect the entity state behind the interface. BIP executor is using Akka Typed Actors to have the benefits of one message processing at a time (no shared state between multiple threads), but it also provides mechanisms to evaluate what transitions (functions of interface are available at a given state).

A level of abstraction (detail) in the definition of FSM is chosen by the developer. For example, a simple FSM consisting of states on and off may be more than sufficient to allow proper control of the entities.

FSM is a necessary approach to explicitly state the different states the component can be with all different benefits coming from this explicit statement.

Coordination

IT infrastructure is a large number of collaborating entities. For the whole IT infrastructure to function properly the constraints of different elements must be respected. Therefore, there is always a need for coordination where one entity agrees to perform a given functionality on behalf of others. The fact that a message broker is in a state that does not allow accepting new messages will force other components of that system to act differently.

The interactions between different components of IT systems should not pollute the implementation of those IT components. The collaboration/coordination (synchronous transitions) is extracted from those components, represented separately as BIP glue and enforced by BIP engine that decides based on this specification how IT infrastructure as a whole should act.

The complete picture is where asynchronous and synchronous actions can take place. BIP allows for both. For asynchronous it uses the same approach as Akka, for synchronous it proposes BIP glue formalism to define a model of synchronous atomic interactions that can happen between the elements of the system.

Behaviour Model

In the context of CF/BIP integration, BIP components can only interact with controlled processes (routes) through

existing interfaces, as opposed to generating the complete process code by the BIP tool-chain. Hence, some events can happen spontaneously, without any possibility of control by BIP components. As it can be seen in the Use case, such spontaneous events can influence the state and the actions that must be taken on the corresponding components. This means that these events have to be observed and reflected by transitions in the behaviour of the corresponding components. To address this issue, we extend the definition of BIP behaviour with by typing all ports as either spontaneous or enforceable. The former correspond to observable but uncontrollable events, while the latter represent the usual BIP ports that can participate in the interactions controlled by the BIP engine.

Basic behaviour for BIP components

Definition (Basic behaviour): A basic behaviour is a Labelled Transition System defined given by a quadruple B = (Q, P_e, P_s, -->), where

- Q is the set of states,
- P_e is the set of enforceable ports,
- P_s is the set of spontaneous ports,
- -->, a subset of Q x P x Q with P = P_e U P_s U {ε}, is the transition relation satisfying the condition below.

Denoting E(s) = { (s,p,s') in --> with p in P_e}, S(s) = { (s,p,s') in --> with p in P_s} and I(s) = { (s,ε,s') in -->} respectively the sets of enforceable, spontaneous and internal transitions from state s in Q, the transition relation --> must satisfy the following condition: for any state s in Q, at most one of the sets E(s), S(s) and I(s) is not empty and, for any p in P, there is at most one outgoing transition from s labelled p. In other words, we require that, from any state, there be transitions of at most one type, among which no two have the same label.

Execution semantics for the basic behaviour

The execution semantics of a basic behaviour is informally[2] defined as follows. A BIP component encapsulating a given behaviour B = (Q, P_e, P_s, -->) is executed in an environment consisting of a BIP Engine and a number of Notifiers. The BIP Engine has the information about other BIP components present in the system and all the associated coordination constraints. Each Notifier observes one spontaneous event generated by the controlled processes and associated to a port from the set P_s and signals the occurrences of this event to the BIP component. Thus, execution of a BIP component follows the cyclic protocol below:

1. In a given current state s in Q, the component computes the sets E(s), S(s) and I(s). Only one of these sets is not empty.
2. If I(s) is not empty, it contains a unique transition (s,ε,s'). The current state is immediately set to s' and the execution cycle restarts from step 1.
3. If I(s) is empty,
    a. if S(s) is not empty, the component waits for an occurrence of one of the spontaneous events labelling the transitions in S(s).
    b. if E(s) is not empty, the component sends the list of enabled ports to the BIP Engine and waits for the Engine's choice of the transition to take.
4. Depending on the choice in step 3, the BIP Engine or one of the Notifiers notifies the component that a transition must be taken, labelled by a port p (belonging to P_e or P_s respectively).
5. There exists a unique transition (s, p, s'). The current state is immediately set to s' and the execution cycle restarts from step 1.

Guarded transitions

Although the basic behaviour defined in Section 3.1 provides a formal model for BIP components in the CF/BIP

integration context, it is not directly amenable for practical use. Indeed, this model requires the state space to encapsulate all relevant information about the entire system, thus enforcing a rather low abstraction level. For instance, consider the component in Figure 1 (see also Section 4). When the current state of this component is wait, the stopRoute() method has been invoked on the associated Camel route (transition off). Depending on the current state of the route the BIP component must either immediately take an internal transition to the state done (if the route has finished all data processing) or wait until an end notification (if the last data chunk is still being processed). Thus, two outgoing transitions — one internal and one spontaneous — must be possible from the state wait, contradicting the condition in the basic behaviour definition.

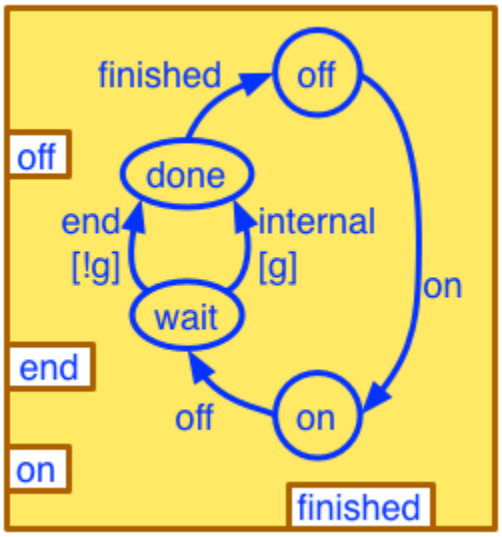

Figure 1. BIP component modelling the control interface for a Camel route.

While it is possible to model this as a basic behaviour above, such a model would be too cumbersome. It would obscure the important control points beyond unnecessary implementation details. Therefore, we extend the model in a common straightforward manner, by associating to each transition a guard. A guard is a boolean predicate, which determines whether a transition is enabled or disabled. The constant predicate true indicates that a transition is always enabled. In the example of Figure 1, transitions end and internal have the guards !g and g respectively, where the value of g is determined by a call to a method isFinished(). Denoting a transition guarded by a predicate g by (s, p, g, s'), we redefine the sets E(s), S(s) and I(s) to contain only enabled transitions as follows: let s in Q be the current state of the component and σ be the current state of global system and put E(s) = { (s,p,g,s') in --> with p in P_e, such that g(σ) = true } and similarly for S(s) and I(s), where g(σ) is the valuation of the guard in the current global state. The condition in the behaviour definition becomes: "from any state, there are enabled transitio ns of at most one type, among which no two have the same label".

> ⚠ If, from a given state, there are outgoing transitions of different types or labelled by the same port, it is the developer's responsibility to ensure that the corresponding guards are disjoint.

Data variables and actions

In addition to the guards introduced in the previous section, it is often convenient to store local data in a component. Such data can be updated upon transitions. In classical BIP, data variables can be both read and overwritten by the Engine upon an interaction. Here, for simplicity, we restrict this model by imposing an private/visi ble typing on data variables. The following rules of data manipulation have to be respected:

- A variable can be updated only by the component it belongs to.
- A private variable can be read only by the component it belongs to.

- A visible variable can be read by any component.
- All visible variables must be updated by their respective container components before any other components can read them.
- Internal and spontaneous transitions should not access data variables of other components.

In order to update data variables, we associate to each transition an action. An action is method defined in the component, which is called during the execution of the transition. Actions also provide a point where control can be executed by BIP components on the corresponding processes of the system. We denote (s, p, g, f, s') a transition from state s to state s' labelled p, guarded by a predicate g and having an associated action f.

The final protocol defining execution semantics of BIP components follows the cyclic protocol below:

1. In a given current state s in Q, the component
    a. updates its visible variables,
    b. computes the sets E(s), S(s) and I(s). Only one of these sets is not empty.
2. If I(s) is not empty, it contains a unique transition (s,ε,g,f,s'). The action f is executed, the current state is immediately set to s' and the execution cycle restarts from step 1.
3. If I(s) is empty,
    a. if S(s) is not empty, the component waits for an occurrence of one of the spontaneous events labelling the transitions in S(s).
    b. if E(s) is not empty, the component sends the list of enabled ports to the BIP Engine and waits for the Engine's choice of the transition to take.
4. Depending on the choice in step 3, the BIP Engine or one of the Notifiers notifies the component that a transition must be taken, labelled by a port p (belonging to P_e or P_s respectively).
5. There exists a unique transition (s, p, g, f, s'). The action f is executed, the current state is immediately set to s' and the execution cycle restarts from step 1.

Use case

In Connectivity Factory, Camel routes are used to process and transfer data. Since the routes often interact with databases, performance optimisation requires that data are bundled in rather large chunks, going up to several hundred megabytes. Tens or hundreds of Camel routes can be active simultaneously. Testing the entire system is infeasible due to its complexity. Therefore, routes are tested individually, which implies sufficient memory is always available at design time. At run time, simultaneous activation of many routes can exhaust the memory, causing an Out of Memory error, which is difficult to trace. An alternative solution is to provide a mechanism that would enforce a limit on the number of simultaneously active routes, thus providing by-construction protection of memory usage. Such a mechanism is difficult to design ad hoc and would be even more difficult to validate. The design of any route should not be influenced/coupled to the mere existence of other routes that can use memory resource. On the other hand, BIP provides a structured approach to component coordination, addressing the problem in a formally provable way.

In this section, we present the BIP model of the use case. The purpose of this model is to provide an overall understanding of the resulting system from the BIP viewpoint. Hence, except for the BIP engine, which implements BIP coordination semantics, it shows all the involved entities. More precisely, the model consists of four components. Two of these components are abstractions of code provided by developers. Thus they will "disappear" in the final system. The other two are the BIP front-end for the route and the Monitor enforcing the limitation of the number of routes. These two components will be instantiated by two beans based on the BIP Executor class (see short description of coding task - Executor implementation).

The first four subsections present the individual components. The last subsection presents the complete BIP model.

Routes

A Camel route is a software entity responsible for transferring data from the source to the destination of the route. This usually involves some processing. A route can be in active, stopped or suspended mode. In the context of this use case, we will only consider active and suspended modes. An active route monitors its source. When data is available at source, it is processed and delivered to the destination. We assume that the events corresponding to the beginning and the end of data transfer are spontaneous, i.e. they cannot be externally controlled (triggered or prevented). However, they can be observed. A suspended route can be activated by a call to the function resumeRo ute() provided by the Camel context. Similarly an active route can be suspended by a call to the function suspendRo ute(). We assume that, when the route is not processing any data, it can be suspended immediately. When suspend Route() is invoked on a route that is processing data, the route is suspended gracefully: it terminates processing the data before switching to the suspended state (see Figure 2 below). If the route implements SuspendableService then it is possible to specify the shut-down strategy.

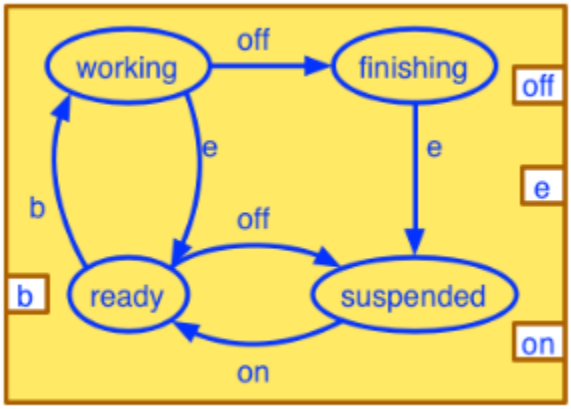

Figure 2. A BIP component (LTS) modelling the behaviour of a Camel route.

In this LTS, b and e are observable, but uncontrollable events corresponding to the beginning and the end of data processing by the route. Events on and off are controllable and associated with the actions of resuming and suspending a route.

This LTS is an abstraction of the real route behaviour, showing only the aspects which are of interest for developers and users in this use case. A more detailed model might be necessary for more advanced control. This LTS must be provided by a developer with Camel expertise. It is useful for designing and validating the associated BIP Control specification (see Section 4.2 below) used for the external control of the route.

BIP Control

BIP Control is a component that enforces coherent control of the route. It must also be defined by a developer with Camel expertise and can be viewed as a protocol that must be respected by the external control when interacting with the route. The BIP Control for Camel routes designed for this use case is shown in Figure 3.

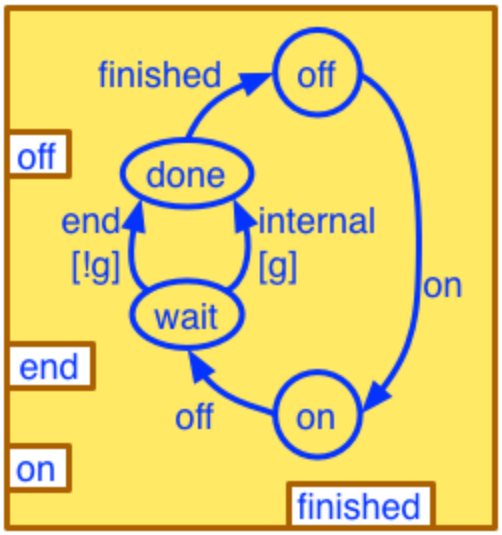

Figure 3. BIP component (LTS) modelling the control interface for a Camel route.

The BIP Control above is in the off state when the route is known to be suspended. Similarly, it is in the on state, when the route is known to be active. The transition from the state off to the state on labelled on has, as the associated action, a call to resumeRoute(). Similarly, the off transition from state on to state wait has, as the associated action, a call to suspendRoute(). Since the timing of the reaction to this call depends on the current state of the route (ready or working in Figure 2), the BIP Control has to ensure that the route is actually not processing any exchange in flight before it transits to the state off. To this purpose the BIP Control calls its own function isFinis hed() and stores the result in a boolean variable g. (This function must be implemented by a developer with Camel expertise.) If the route is not processing any data (g == true), the BIP Control makes an internal (immediate) transition to the state done. If the route is processing data (g == false), the BIP Control waits for a notification about the spontaneous end transition. Finally, in the state done, it is known that the route has been stopped or suspended and the BIP Control switches to the state off. The transition finished is useful for signalling the fact that the route has been stopped or suspended to other components, e.g. the BIP Monitor (Section 4.4).

Notifier

Beginning and end of data processing by a Camel route are not externally controllable, but can be observed. Other components – BIP Control, in this use case, – have to be notified about these events so that they can take corresponding spontaneous transitions (here, the transition end from wait to done in the BIP Control of Figure 3). To this end, a Notifier entity is created that has a sole responsibility: to monitor the route and inform the concerned components when the associated event is produced. In the current model of the use case, we are only interested in the end event, corresponding to transitions e in the route model (Figure 2). Therefore the BIP model of a notifier is trivial (Figure 4).

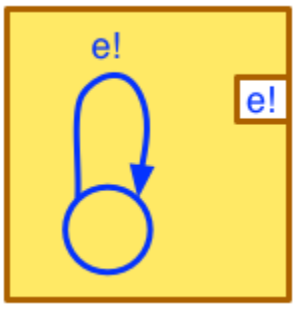

Figure 4. BIP component (LTS) modelling the notifier associated to the event of ending data processing by the route.

BIP Monitor

A BIP Monitor is a component to be added to the system by a developer or a CF Administrator, which enforces satisfaction of required safety properties. In this use case, we require that no more than two controlled routes are active simultaneously.

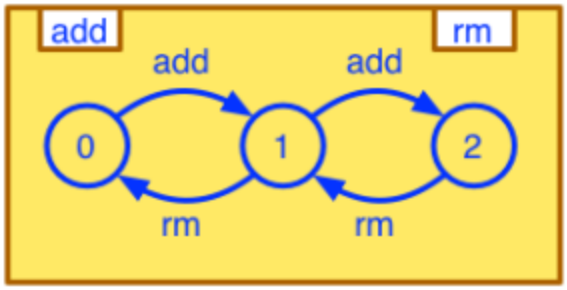

Figure 5. BIP component (LTS) modelling the BIP Monitor ensuring that no more than two controlled routes are active simultaneously.

The component in Figure 5 is a simple counter, which has three states reflecting the number of active routes. When a controlled route is switched on or suspended the Monitor is notified through the ports add and rm (remove) respectively. In the state 2, when two routes are active simultaneously, the add port is disabled (since there is no corresponding transition), thus blocking the activation of new routes. This is enforced by the synchronisations defined in the complete BIP model in Section 4.5.

BIP model of the use case

The BIP model in Figure 6 shows the connectors defining synchronisations between the components presented in Sections 4.1–4.4. It should be observed, that this figure only shows one Camel route, while the use case assumes the existence of several. Therefore a complete model would have one copy of each of the route behaviour, Notifier, BIP Control and all the corresponding connectors per controlled route.

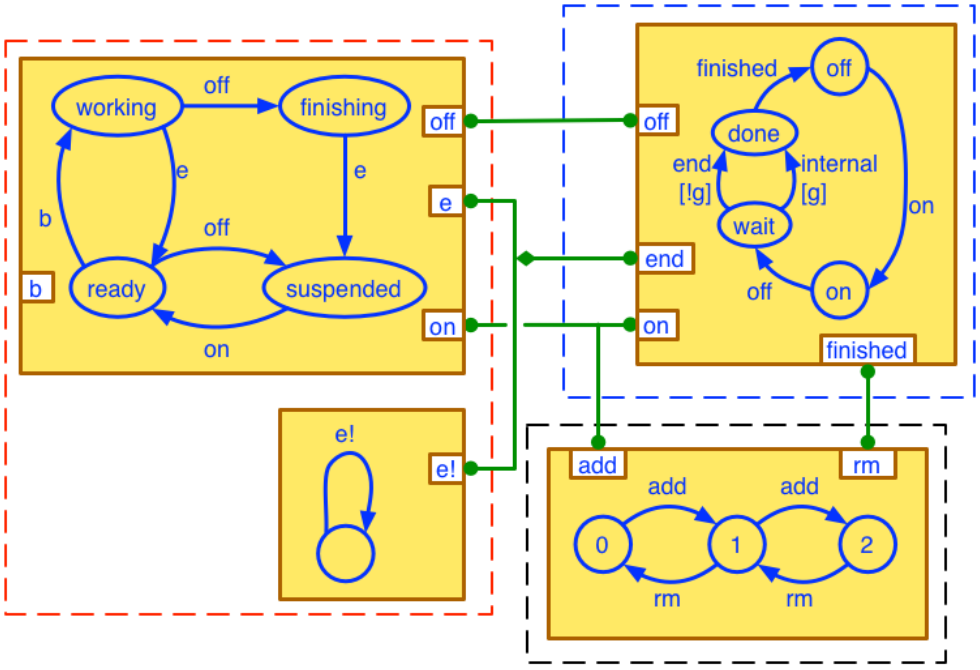

Figure 6. BIP model of the use case showing one route.

Here, off-off, finished-rm and on-on-add are respectively binary and ternary strong synchronisations with obvious purpose. The remaining connector is structured: it has a strong binary synchronisation sub-connector e-e! (the notifier observes the e transitions of the route behaviour and generates the corresponding notification event). This sub-connector is then a trigger (BIP terminology) for the binary connection with the end port of the BIP Control. According to the BIP semantics (Algebra of Connectors), this means that the interaction e-e! can happen on its own, whereas the transition end of the BIP Control will participate when it is available. This accounts for the situation, where the route is processing data and no suspend request (call to the suspendRoute() function) has been issued. In this case, the BIP Control is in the on state, where the transition end is not available and the corresponding port is disabled. Since the synchronisation is not strong (presence of the trigger in the connector), the e transition of the route is not blocked.

The components in the red box are abstractions of software entities that are not controlled by the BIP Engine. Route behaviour model is just an abstraction of existing software entities, whereas Notifier represents a separate entity, which will be created without an explicit BIP model. BIP Control and Monitor, on the other hand, are BIP modules, which will be defined by their BIP models and executed by a BIP Model Executor bean to be developed separately (only once – see short description of coding task Executor implementation). The BIP Model Executor instances will interact with the BIP Engine to ensure correct coordination. The blue box contains the modules (here only BIP Control, but potentially also others?) which interact directly with the controlled processes. The black box contains additional modules (here only BIP Monitor, but again...) which impose control policies. These latter modules do not interact directly with controlled processes.

Software architecture

BIP API

BIP specification provides the information about FSM. Any BIP component must have a specification that is provided as a class. An instance of the specification is actually a model of the particular instance of BIP component. E.g. a SwitchableRoute class is a specification of the SwitchableRoute component. An instance of this specification is modelling particular instance of the route itself that must be switchable.

BIP Spec can be any POJO, in particular it can be a Spring bean. Thus, BIP spec can implement interfaces of other frameworks making it usable within frameworks like Spring. Functions within BIP spec can therefore use anything that was injected by Spring. There is no limit to the nature of the code that will be executed upon transitions from one state to another state.

A specification contains information provided by developer to specify FSM of the component and BIP framework will enforce that this FSM specification is respected.

BIPActor

BIPActor contains the minimum functionality of BIP Component that can be used by the "environment" to send spontaneous events to entities being coordinated by BIP engine. There are two inform functions. The first one just informs that a particular event has occurred and this event does not carry any event specific data. Therefore, the inform has only one parameter of type String and name portID to indicate what spontaneous port of BIP component is receiving the spontaneous event. The second inform function makes it possible to send any event specific data with the help of the map. Every piece of data is indicated by its name (key of the map) and has a value (value associated to the key). This map contains data that will be mapped against the signature of the function that handles spontaneous events within BIP spec.

```
public interface BIPActor {
    /**
     * It informs BIP component that a given spontaneous event associated with a
port id has
     * occurred.
     *
     * @param portID
     *            the port id specifying a spontaneous event that has occurred.
     */
    void inform(String portID);
    /**
     * It informs BIP component that a given spontaneous event associated with a
port id has
     * occurred.
     *
     * @param portID
     *            the port id specifying a spontaneous event that has occurred.
     * @param data
     *            the data provided with a spontaneous event.
     */
    void inform(String portID, Map<String, Object> data);
}
```

BIPComponent API

It is the API provided by BIP-component bundle that publishes itself as an OSGi service. It has three functions. First notice that it extends interface Identifiable and BIPActor. BIP component adds extra functions that are associated to the interactions with BIP framework internal entites namely Executor and BIP engine. Function execute will be called by BIP engine to force the execution of the enforceable transition. Function getData and checkEnabledness will be called by Executor that collects information about the current state of BIP component and enabled enforceable transitions to inform BIP engine that makes decision which enforceable transition should be executed. Function setData provides possibility to specify the data that are required to execute the enforceable transition. This interface should not be used directly by developer as it contains blocking calls and only is used internally within BIP entities.

```
ppublic interface BIPComponent extends Identifiable, BIPActor {

    /**
     * It enforces the execution of a given enforceable transition associated to a
given port id.
     *
     * @param portID the port id of the transition being enforced.
     */
    void execute(String portID);
    /**
     * It provides data specified by the name parameter casted to the type provided
by clazz parameter.
     *
     * @param <T> the generic type
     * @param name the name of the data being requested.
     * @param clazz clazz specifies the type of the data being returned.
     * @return the data that was requested as specified by name and clazz parameter.
     */
    public <T> T getData(String name, Class<T> clazz);
    /**
     * It makes it possible to query BIP Component if a given port is enabled for
given data parameters.
     *
     * @param port the port of the associated transition being checked for
enabledness.
     * @param data specifies a list of data parameters, where each map is a complete
set of data needed by the transition.
     * @return the list of booleans specifying for each map if the transition is
enabled or not.
     */
    public List<Boolean> checkEnabledness(PortBase port, List<Map<String, Object>>
data);
    /**
     * Provides data to a BIP component for a given data dependent transition that
will be soon enforced.
     *
     * @param dataName it specifies the name of the data being specified.
     * @param value it specifies the value of the data identified by dataName
parameter.
     */
    public void setData(String dataName, Object value);
}
```

BIP-engine API

It is an API provided by BIP-engine bundle. It allows to inform BIP-engine about stable states reached by the BIP component. Functions inform are used internally by BIP entities thus are of no concern to the BIP user. In general, components needs to be registered first followed by BIPGlue specification followed by start() and execute(). The stop function should be only used when we no longer want BIP engine to coordinate collaboration between BIP components.

```
public interface BIPEngine {
    /**
```

```
     * It specify the BIP glue to be enforced by BIP engine when choosing
interactions.
     *
     * @param glue the BIP glue
     */
    void specifyGlue(BIPGlue glue);

    /**
     * It register a given component with its behavior within the engine to be
managed by the engine.
     *
     * @param component the component
     * @param behaviour the behaviour
     */
    BIPActor register(Object component, String id, boolean useSpec);
    // It only specifies the disabled ports with no data transfers, some ports with
data
    // transfers may have disabled status but this has to be obtain by querying
BIPexecutor with the help
    // of checkEnabledness. If there are no for certain ports that are disabled then
this array list is empty.
    // A port that requires data transfer may still be disabled no matter what data
transfer occurs that can
    // be already indicated here, so the ports specified within disabledPorts may
require data transfers.
    /**
     * It informs BIP engine about the state of the BIP component.
     *
     * @param component the component for which the information about the state is
provided.
     * @param currentState the current state of the component for which the
information is provided.
     * @param disabledPorts the disabled ports of the component for which the
information is provided.
     */
    void inform(BIPComponent component, String currentState, Set<Port>
disabledPorts);
    /**
     * InformSpecific is called multiple times for each component instance. It
allows for the deciding
     * component to specify for each deciding port the disabled combinations for
each potential collaborator
     * of deciding component.
     *
     * @param decidingComponent the deciding component
     * @param decidingPort the deciding port
     * @param disabledCombinations the disabled combinations
     */
    void informSpecific(BIPComponent decidingComponent, Port decidingPort,
Map<BIPComponent, Set<Port>> disabledCombinations);
    /**
     * It starts the BIP engine thread.
     */
    void start();
    /**
     * It stops the BIP engine thread.
     */
    void stop();
    /**
```

```
     * It starts the execution of BIP engine cycles for the registered BIP
components.
     */
```

```
    void execute();

}
```

BIPActorAware

It may be required that BIP spec needs to send reference to itself to another BIP component so spontaneous events can be received. The BIPSpec object must implement BIPActorAware making BIP Engine inject proper proxy of BIP spec that implements this interface.

```
public interface BIPActorAware {
    void setBIPActor(BIPActor actor);
}
```

BIP annotations needed for BIP Specification

BIP specification is provided within a class. The instance of this class will be provided as the input to BIP executor to enforce FSM described within BIP spec.

ComponentType

The annotation annotates the whole class. This annotation serves two purposes. First, it allows to specify the type of the component with the help of name attribute. This type information will be used/present in BIP Glue specification to specify the coordination logic. Second, the attribute initial allows to specify the state the BIP component enters upon creation through any constructor.

```
@ComponentType(initial = "initial", name = "org.bip.spec.seal.SealableData")
public class ... {
}
```

Ports & Port

This annotation annotates the whole class. This annotation has two attributes. The first one, name, specifies name of the port. The second one, type, specifies the type of the port. The are two possible types of ports. The enforceable port and the spontaneous port. Ports annotation is just to be able to annotate the class with multiple Port annotations. Each port will have a backing of the transition within a component.

```
@Ports({@Port(name = "end", type = PortType.spontaneous),
               @Port(name = "on", type = PortType.enforceable),
               @Port(name = "off", type = PortType.enforceable),
               @Port(name = "finished", type = PortType.enforceable) })
```

Transitions & Transition

Transitions annotation is just to be able to assign multiple transitions to the same function. Transition annotation has four attributes. The first one name must correspond to the existing Port name of the BIP component. The type of the port with the same name as the transition will indicate if the transition is enforceable or spontaneous. If the name is an empty String then it indicates that this transition is internal and not visible to the outside world. The internal transitions are provided as important convenience method to be able to specify desired FSM. The attribute source and target define respectively the source state of the transition and the target state of the transition. Finally, the transition of any type may not always be enabled and the guard expression is provided as the guard attribute.

```
@Transition(name = "off", source = "on", target = "wait", guard = "")
public void stopRoute() throws Exception {
   logger.debug("Stop transition handler for {} is being executed.", routeId);
   camelContext.suspendRoute(routeId);
}
```

The executor knows in what state the BIP component is. Therefore, the executor with the help of the guards will check what transition are enabled.

Guard

Guard annotation is to annotate a function returning a boolean that can be used to compute the condition that may be used in the guard expression. Please note as guard computation can use anything that is available within a BIP spec. Moreover, the guard may require some data that is provided by the participating component with the help of the data wire. Guard should have no side-effects that can influence the state the BIP spec is. No guarantess of correctness of BIP executor can hold if guard functions have side effects.

```
@Guard(name = "isFinished")
public boolean isFinished() {
    return camelContext.getInflightRepository().size(routeId) == 0;
}
```

ExecutableBehaviour

Annotations to specify FSM of BIP component will be used to create a BehaviourBuilder. It is possible to provide your own function within BIP spec that returns BehaviourBuilder and is annotated with @ExecutableBehaviour annotation to have possibility to programmatically specify FSM. Please note that you will still can use Data annotation to specify Data transfers functionality within BIP spec class.

```
@ExecutableBehaviour
public BehaviourBuilder ...() {}
```

Data

Data annotation can appear in different contexts as specified below. The attribute name is important for all of the contexts.

Output data

This context uses two additional attributes, accessTypePort attribute to specify the semantics of the another attribute called ports. The Data annotation is used in this context for the function that returns a particular type. In the example, below it is actually a generic type. BIP executor is caching the result of this function for each BIP cycle execution, thus there is some protection if this function is not idempotent. However, there is no possibility to protect against side effects being done by this function.

```
@Data(name = "value", accessTypePort = AccessType.allowed, ports = {"get"})
public T get() {
    return data;
}
```

The possible values for enum accessTypePort are specified below.

```
/**
     * The Enum PortType.
     */
    public enum AccessType {
        /** The any specifies that data is available if any port is participating in
the interaction. */
        any,
        /** The witness specifies that data is available even if no port is
participating in the interaction. */
        witness,
        /** The allowed specifies that data is available only for the ports that are
specified. */
        allowed,
        /** The unallowed specifies that data is *not* available for the ports that
are specified. */
        unallowed
    }
```

Input Data

Here is an example of the input data that is required to perform transition set. This input data will need to be provided by the other component that is participating in the interaction and will be specifed by BIP glue data wire specification.

```
@Transitions( { @Transition(name = "set", source = "set", target = "set", guard =
"!isSealed"),
                @Transition(name = "set", source = "initial", target = "set") } )
public void set(@Data(name = "input") T data) {

   logger.debug("Previous data {}, Current data: {}", this.data, data);
   this.data = data;
   noOfTransitions++;

}
```

Guard Data

Here is an example of the guard function that may require to receive data to decide if the transition is enabled or not. The example below is taken from HanoiTower example where the transition addPiece has a guard that expects to receive information what is the size of the disk that is being added to the peg. Please note that the same data (addedDisk) will be used later on when executing transition addPiece. The same data (value) will be used for the guard calculation and transition execution. No guarantees are provided that it will be the same instance in case of data being non primitive type.

```
public boolean isPieceAddable(@Data(name = "addedDisk") int no) {
     if (no == -1)
          return false;
     if (no == 1)
          return !pieces[0];
     for (int i = 0; i < no - 1; i++) {
          if (pieces[i])
             return false;
     }
     return !pieces[no - 1];
}
```

BIP Modules

For optimal modularity, the implementation of a BIP component — a BIP Module — is split into two classes:

- BIP component specification — a class with annotations to define ports, states, transitions, and guards within the component;
- BIP Model Executor — a generic class, referencing a BIP component specification and implementing the execution semantics defined in the previous section.

Spontaneous events can occur at any moment during the system execution, independently of the BIP component state. BIP components should be able to perform simple operations, such as logging or counting, when these events occur. This is addressed by separating event processing into two groups of methods: port and event handlers.

A BIP component specification is a Java class defining a BIP component type through BIP-specific annotations. These annotations have to identify all syntactic elements composing a BIP component: component type name, ports, states, transitions, data variables and guard predicates. Below we provide the list of such annotations with their corresponding parameters.

- @bipComponentType — associated to a BIP component specification class
  - name — the name of the component type
  - initial — the name of the initial state
- @bipPort — associated to a port handler method,
  - name — port name
  - type — "spontaneous" or "enforceable"
- @bipTransition — associated to a transition handler method
  - label — the name of the port labelling the transition
  - source — source state
  - target — target state
  - guard — name of the method to be invoked to evaluate the guard of the transition (true if omitted)
- @bipGuard — associated to a method used as a transition guard, can be used to statically verify the signature of the method, which should take no arguments and return boolean

- @bipData — associated to a property
  - access — in or out
  - name — for out-variables, the name by which the data variable should be referred to by other components
  - type — type of the data

The BIP Model Executor is a generic class (or bean), implementing the execution semantics of BIP component behaviour. It must reference the corresponding BIP component specification, keep track of the current component state and provide an interface for interaction with the environment (BIP Engine and Notifiers). Furthermore, through reflection, BIP Executor should implement an interface giving access to component information such as component id, list of ports, list of enabled ports, list of visible variables.

The software architecture proposed for CF/BIP integration is shown in Figure 7. Elements shown in light blue are Java beans, in light green — BIP Modules, in grey — OSGi bundles. Arrows shown in blue represent direct method calls, those in black represent calls to services through OSGi-established links.

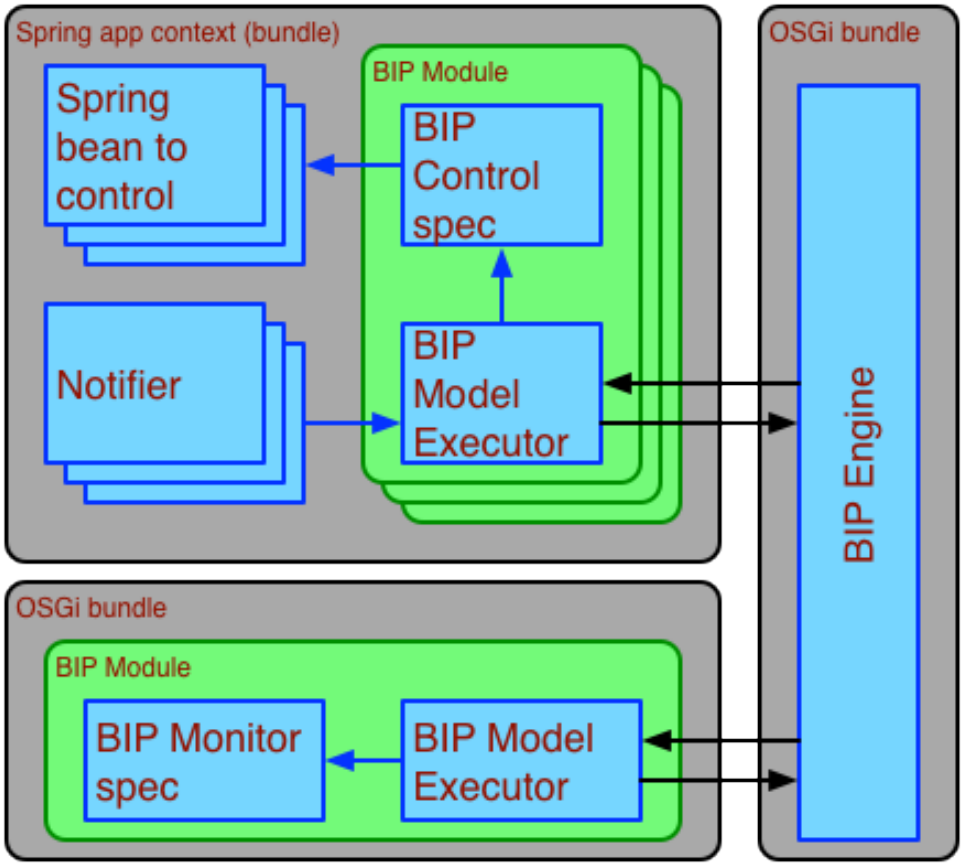

Figure 7. Proposed architecture for the CF/BIP integration.

Each bundle in a controlled system can consist of several beans. A developer with sufficient knowledge of the dynamics of these beans defines one or several BIP component specifications for control modules. Although, typically, there would be one BIP Control module per controlled bean, there is no a priori restriction: any number of BIP Control modules can collaborate to control any number of beans. Each instance of BIP Control specification is coupled with a dedicated instance of BIP Model Executor to form a module. Instantiation and dependency injections are managed by the Spring context of the bundle. In addition, one or several (only one in the figure) BIP modules can be added to the system in order to monitor the entire system and impose control policies to satisfy safety requirements. Such monitor modules can — but do not have to — be packaged as a separate bundle, for example, when provided by CF administrators (users) rather than by developers. Execution of the entire control layer, i.e. the

complete set of BIP modules, is driven by a BIP Engine. This latter has to provide a service for BIP modules to register for coordination. In addition, service discovery can be implemented by the BIP Engine in order to automatically detect new BIP components.

Glossary

BIP specific

Labelled Transition Systems (LTS)

Labelled Transition System (LTS) is a mathematical (i.e. abstract) notion used to model execution dynamics of systems having a set of identifiable states and evolving from one state to another in discrete steps. In the case of computer programs, the canonical view is to consider all possible memory assignments (including both data and control parts) as states. Each program instruction defines a transition between two states. Even when such an instruction does not update any variables, its execution still results in the modification of the program counter. Depending on the particular modelling objective, some information (memory regions) can be disregarded or abstract ed away in the model, thus reducing the state space. Although it is important, when modelling a system, to choose the right abstraction level, the same mathematical structure, i.e. LTS, is used to represent the model.

An LTS is defined as a triple (Q, A, -->), where

- Q is the set of states,
- A is the set of labels,
- -->, a subset of Q x A x Q, is the set of transitions.

An element (s, a, s') of -->, also written s --a–> s', is a transition from state s to state s' labelled a. Commonly, one also explicitly defines the initial state(s) of an LTS. Here, we omit this part for the sake of conciseness.

Often extensions of LTS are considered to simplify the discourse. For instance, one can associate to LTS data variables of discrete types (boolean, integer, etc., but not real). In this case, in addition to labels, transitions are annotated with guards – boolean predicates on variables – and actions. Transition is said to be enabled in a given state with a given valuation of data variables if the guard predicate evaluates to true. Such a transition can then be executed, whereby, in addition to switching to the target state, the associated action of the transition is used to update the data variables. Such extensions are equivalent to the definition above in terms of expressive power, since valuations of discrete data variables can be incorporated into the state space.

Other extensions of higher expressive power can be considered, such as Timed or Hybrid Automata, which can have associated variables evolving continuously. Such automata are used to model model physical processes involving parameters such as time, temperature, etc.

Separation of concerns

BIP is based on the following separation of concerns principle: in order to tame the system complexity inherent to concurrent applications, we separate sequential computation (also called behaviour) and coordination. Necessarily, the coordination mechanisms must not perform any computation, which means that they do not involve any kind of persistent state or memory that can be modified by coordination per se. Thus, coordination is memoryless and all computation is embodied in atomic components.

BIP Component and Ports

In BIP, systems are built up from atomic components using glue. The distinction between the two notions is due to the separation of concerns principle explained above. In order to further reduce the complexity, the system can be

structured by assembling components together to form compound components. The formal definition of the operational semantics of glue in BIP allows one to discard the structure of compound components and consider them semantically as equivalent atomic ones. This observation also lies at the heart of several model transformations used as part of the BIP design flow.

A component in BIP is modelled by an LTS (see above). Labels of the LTS are called ports. They provide the interface through which the component can interact (synchronise) with other components. For atomic components, the LTS must be given explicitly. For compound components, the LTS are implicit and represent the semantics of a component as mentioned above. Although variants of BIP exist, which use Petri Nets or Timed Automata to model component behaviour, these are irrelevant for the project in its current set up.

In classical BIP, the LTS specification of an atomic component is used to generate the code implementing this component in the final system. In our project, we take the opposite approach and mostly use BIP components as abstract models of existing processes running in CF. Some exceptions remain, however (see the BIP Monitor below).

Glue: Interactions and Priorities

Glue is a mechanism (operator) defining coordination (see Separation of concerns). In BIP, there are two types of glue operators: Interaction and Priority models.

Interaction model is a set of allowed interactions. From the abstract model point of view, an interaction represents simultaneous execution of a set of transitions from different components. Thus an interaction is a set of ports.

According to BIP operational semantics, at each execution round only one interaction can be executed. However, several interactions can be enabled. In such a case, the BIP Engine (see below) non-deterministically picks one enabled interaction for execution. This non-determinism can be reduce by assigning to some interactions higher priority over some other ones. Thus a priority model is a strict partial order on the interaction model.

One can say that interactions define collaboration between components, whereas priorities define the conflict resolution policies. Guarded priorities, i.e. priorities activated only when a given boolean predicate on the data variables evaluates to true, can be used to enforce scheduling policies such as Round Robin, Earliest Deadline First (EDF), etc. or mutual exclusion.

Several possibilities exist for specifying interaction models, among which

- Connectors, formalised by the Algebra of Connectors, based on two basic protocols: strong synchronisation and broadcast. For the latter, a port typed as a trigger can initiate an interaction wherein other ports (commonly typed as synchrons) can participate.
- Boolean macros, formalised by an algebra of boolean constraints. This approach is used in Dynamic BIP, where each transition has its associated coordination constraints expressed with two macros: Require and Ac cept defining the interaction the transition can participate in.

Although, in the project, we will use boolean macros to define interactions, connectors are useful to graphically represent the models.

Engine

A BIP Engine is a software entity that implements the operational semantics of the BIP glue. Execution of a BIP system follows the protocol below:

At each execution cycle

1. Atomic components reach their respective stable states and notify the Engine, providing the list of enabled transitions (ports);

2. The Engine computes the set of enabled interactions of highest priority (i.e. not dominated in the priority model by any of the enabled interactions);
3. The Engine picks one of the computed interactions non-deterministically, executes any associated data exchange operations and notifies the involved components, providing the names of the respective ports;
4. Each component involved in the selected interaction executes the corresponding transition. Components not involved in the selected interaction do nothing.

In step 3, non-deterministically means that the manner in which the Engine picks one interaction is not specified. Depending on the implementation context, this can be the first interaction in the list, a random interaction, an interaction selected by a human operator, etc.

Safety and Liveness Properties

Safety and liveness properties are the two kinds of requirements commonly imposed on systems. Intuitively, safety properties require that something bad never happens, whereas liveness properties require that something good eve ntually does happen. Thus, safety properties involve statements about the past history of the system execution and its current state, whereas liveness properties refer to the future of the execution. Therefore safety properties are easier to monitor and impose, in particular by blocking transitions that would violate a given safety requirement. This observation also provides a method for reducing any safety property to freedom from deadlocks. Indeed, if a system extended with such a monitor can be shown deadlock-free, this automatically implies that the safety property is satisfied.

BIP Monitor

A BIP Monitor is an additional BIP component, which does not model any pre-existing CF processes, but enforces upon the system a policy implementing required safety properties. A system can have any number (0,1,...) of BIP Monitors. Each BIP Monitor is implemented by a corresponding BIP Module.

BIP Module

A BIP Module is an implementation of an atomic BIP component in the architecture proposed in the project. It consists of

1. An annotated Java class specifying the component behaviour as an LTS;
2. A generic BIP Executor bean, which manages the state of the component, interactions with the BIP Engine and the notifiers.

OSGi specific

OSGi provides proper dynamic modularity mechanism, version control of code, service layer, and 24/7 operation possibility as code can dynamically added or removed.

OSGi container

An OSGi container is an implementation of the OSGi standard. It manages the life-cycle (installation, dependency resolution, activation, etc.) of modules called OSGi bundles. An important part of OSGi container is a registry of the services provided by bundles. The biggest contribution of OSGi standard is the approach for module management, including versioning, depedency resolution, and starting and stopping modules while OSGi container is running. Connectivity Factory is an OSGi container.

Bundle

OSGi bundle is a unit software module that can be managed by an OSGi container. It is a jar file whereof the manifest contains the following additional entries:

- exported code
- code that needs to be imported
- version information

Interface-based design approach encouraged by OSGi consists in isolating all the information necessary to establish connections between different objects in one or several interfaces implemented by the involved entities. Thus actual implementations of these interfaces do not impact the connectivity and can be changed without requiring any system-wide redesign.

OSGi registry

An OSGi registry contains a list of services that are published by the active bundles in the OSGi container. This registry can be queried by any process within the OSGi container to get access to the service implementations.

OSGi service

An OSGi service is a functionality provided by an OSGi bundle, which can be used by other bundles. The best practice consists in declaring the interface implemented by the service separately from the actual implementation. OSGi service can have additional properties that can be used upon querying the OSGI registry to select the appropriate implementation of a service.

OSGi service property

It is simply a key value pair. It can be used when asking for the OSGi service.

Spring

Spring Application Context

It is an entity within which all objects that constitute the system are created. The application context is provided as an XML specification that is being used to instantiate and wire all the objects in the proper fashion. Most of the OSGi bundles written by CT will have Spring context within a bundle.

Spring bean

Spring bean is any of the objects that have been defined within Spring Application Context. Spring bean is simply an instance of the class as specified within bean declaration. Moreover, this bean may reference another bean through an attribute. The dependency (another bean) is then injected into a bean.

Dependency Injection

Spring is a dependency injection framework. It wires the dependencies. The fragile connectivity between objects is outsourced to Spring and there is one main point of control - Spring Application Context.

Camel

Camel is a route mediation engine. It is the heart of CF as it allows to execute the integration flows. Camel receives specification of routes (integration work) and executes those specifications. It is highly modular and easy to extend.

Camel Context

CamelContext is the manager type entity. It is the entity that holds all the routes that must be executed. It provides an API that can be used to control how those route specifications are being executed. Each bundle that is doing integration will have a CamelContext instantiated within a SpringContext. CamelContext manages execution of workflows.

Camel route

It is a simple specification explaining how information is taken from one place to another. An example of a route will be a filemover route :

```
from("file:"+ inFolder +"?delete=true").routeId("cfa.filemover").to("file:" +
outFolder);
```

Route is a core concept in the engine that is used as a definition for routing engine. The simple route example can be specified by from endpoint and to endpoint. Another simple route will just specify from endpoint and processor that processes the messages produced by from endpoint. Usually, a route specifies a processor that is invoked to process the message traveling between from and to endpoints. The more complex routes can be seen as a chain of endpoints through which the message is transported and acted upon by the processors. Each route has an unique id that can be used for logging, monitoring, and debugging. All the routes definitions start with the definition of the input endpoint (from endpoint).

The routes allow to :

- dynamically decide what server will be invoked to handle client request
- add extra processing performed on the message being transported
- independently develop clients and servers as soon as communication is decided upon

Endpoint

An endpoint is an entity that defines an end point that can be used as sending or receiving point. In the picture above the endpoints are represented by factories. Any endpoint is defined by URI (Uniform Resource Identifier). Endpoint URIs consists of three parts: a scheme, a context path, and options. An example of the endpoint URI is "file:/user/home?delay=5000". The colon delimits the scheme (file) from the context path (/user/home). The question mark delimits the context path from options (delay=5000). An endpoint creates multiple entities, namely Exchanges, Producer and Consumer entities.

Exchange

The message is contained during routing in an exchange. An exchange provides support for different interactions between systems. The systems can interact using different message exchange patterns (MEPs). Routing engine differentiates between one-way and request-response communication styles. An exchange has an attribute to specify if it is inOnly or InOut exchange. InOnly property specifies a one-way message. InOut exchange consists of request and response message which can be used to to represent HTTP-based transports.

The exchange consists of :

- An unique ID, which if not set explicitly will be generated by the engine.
- MEP attribute to specify if the exchange is InOnly or InOut.
- Exception field that stores any exception that has occurred during routing.

- Properties that should specify global-level information specific to the whole exchange and not just a particular message. Engine will add properties during routing.
- In message that is mandatory.
- Out message that is mandatory only if InOut MEP is used.

Addendum

BIP framework versus Akka

This document is not meant to be an extensive overview of the difference. It is more meant as a collection of interesting remarks on design differences between Akka and BIP framework.

A disclaimer, BIP uses Akka functionality extensively as it recognizes Akka and Actor based programming value. We do not provide as extensive set of features like Akka does. Although BIP could provide some of those features directly thanks to Akka. In BIP we focus on particular aspects that are purposefully (by their design choice influenced by their opinion) ignored by Akka of how the systems should be built. BIP is acknowledging the existence of systems that interact together in a synchronous manner and making it possible to explicitly write down those interactions and enforce them in a proper atomic manner.

Coordination & BIP Glue

BIP is not only about asynchronous messages being passed between entities. BIP assumes that coordination between entities may be required and introduces the concept of atomic interaction. An interaction consists of transitions of multiple BIP components. An interaction must happen in atomic manner, it is not allowed for only part of the interaction to happen.

Akka leaves to poor developers to write coordination protocols between Actors. BIP approach ask the developer or administrator of the system to provide a model of allowed interaction in the system that will be enforced by BIP engine. It is only required to provide the specification of what is proper

FSM

BIP has an opinion that in order to coordinate and manage state of different entities it is required to understand and explicitly state Finite State Machine model for the components. Akka provides possibilities to write FSM based actors but the specification of FSM tend to be cumbersome where high level concepts (state, transitions) are getting intermingled with code associated to those high level concepts. In BIP, with the help of annotation are able to express FSM in an abstract manner. Moreover, it will be relatively easy to move FSM from classes into interfaces extending interface-based design into FSM-based design of systems.

Akka provides become/unbecome functionality, one can see this as an actor transforming itself upon receiving of the message into a different one. Each become definition contains information about state transitions (caused by handling of the received message). Specifying FSM by specifying become for every state of the actor maybe ineefficient way of specifying FSM as the code for each become needs to be written down and can not be reused easily between different states. BIP provides annotation based FSM and Programmatic API based on Reflection.

BIP protocol & BIP component

BIP components can perform internal transition firsts (no need for external messages/stimulus). If no internal transitions allowed then BIP component may perform transitions due to the asynchronous messages it has received (like in Akka Actor). Finally, if no transitions due to external message is possible (no messages, no transitions for received messages) then component can advocate to BIP engine what type of transitions it can do on its own. Each transition that was advertised to BIP engine is a commitment upon which the BIP engine may call.

BIP component not like normal actor can advertise the transitions it can perform in a given state and this commitment may be called upon by BIP engine that makes sure that interaction happening between BIP components reflect the interaction model specified by BIP glue.

BIP Spec

BIP spec which is a FSM specification is just a POJO annotated with BIP annotation. It is the core of the BIP entity (BIP actor, BIP executor). POJO holds all the information needed (state of the BIP component). Moreover, on top of BIP Spec a BIP Executor works to respect FSM-based contract of BIP spec as well as store asynchronous messages that arrive to BIP spec. BIP executor also holds messages received by BIP spec and enforces BIP protocol. Since BIP spec POJO and BIP executor can not die like Akka actor thus there is no problem of loosing state upon exception that in Akka would make an actor die. It gives more control to the developer how to create a POJO a core of BIP component and makes it possible to employ different policies to deal with incoming asynchronous messages. In some sense, one could argue that BIP spec can contain all the business logic without coordination aspect and enforcement of FSM based contract that is explicitly stated by annotation in a manner that does not obscure the business logic.

BIP Presentations

Eclipse Conference 2013 - Technical Summary/Abstract

When building large concurrent systems, one of the key difficulties lies in coordinating component behaviour and, in particular, concurrent access to resources. Native mechanisms such as, for instance, locks, semaphores and monitors allow developers to address these issues. However, such solutions are complex to design, debug and maintain. Indeed, coordination primitives are mixed up with the functional code, forcing developers to keep in mind both aspects simultaneously. Finally, in concurrent environments, it is difficult to envision all possible execution scenarios, making it hard to avoid common problems such as race conditions.

The coordination problem above calls for a solution that would allow developers to think on a higher abstraction level, separating functional and coordination aspects of the system behaviour. For instance, one such solution is the AKKA library implementing the Actor model. An actor is a component that communicates with other components by sending and receiving messages. The processing of a message by an actor is atomic. The state of an actor cannot be directly accessed by other actors, avoiding such common problems as data races. However, component coordination and resource management are still difficult. Fairly complex message exchange protocols have to be designed, which are still spread out across multiple actors. Any modification of the coordination policy calls for the corresponding modifications in the behaviour of several actors, potentially leading to cascading effects and rendering the entire process highly error-prone.

Our approach relies on the observation that the behaviour of a component can be represented as a Finite State Machine (FSM). An FSM has a finite set of states and a finite set of transitions between these states. Transitions are associated to functions, which can be called to force a component to take an action or to react to external events coming from the environment. Such states and transitions usually have intuitive meaning for the developer. Hence, representing components as FSM is a good level of abstraction for reasoning about their behaviour. In our approach, the primitive coordination mechanism is the synchronisation of transitions of several components. This primitive mechanism gives the developers a powerful and flexible tool to manage component coordination. Furthermore, this approach allows us to clearly separate the behaviour of a component and the interface it exposes for the interaction with other components from the system-wide coordination policies.

OSGi associates to each bundle a simple state machine representing the bundle's lifecycle. A bundle can be in one of the states Installed, Resolved, Active, etc. However, once the bundle has been started, it remains in the state Active

— the functional states are not represented. Therefore, this mechanism is not applicable for coordination of active components.

We have implemented functional component coordination in OSGi by using BIP coordination mechanisms (see also the overview on the RiSD web site). BIP is a component framework jointly developed by Verimag (France) and EPFL RiSD (Switzerland) laboratories lead by Prof. Joseph Sifakis (2007 Turing Award laureate). In BIP, systems are constructed by superposing three layers of modelling: Behaviour, Interaction, and Priority. The first layer, behaviour, consists of a set of components modelled by FSM. The second layer models interaction between components defining explicitly which transitions could be synchronised. When several interactions are possible, priorities can be used as a filter. Interaction and Priority layers are collectively called glue. The execution of a BIP system is driven by the BIP Engine applying the following protocol in a cyclic manner:

1. Upon reaching a state, each component notifies the BIP Engine about the possible outgoing transitions;
2. The BIP Engine picks one interaction satisfying the glue specification and notifies all the components involved in it;
3. The notified components execute the functions associated to the corresponding transitions.

To use the transition synchronisation mechanism, developers must ensure that the component states remain stable during one cycle of the above protocol: a component must be able to perform any transition it has announced as possible to the BIP Engine.

As mentioned above, this approach allows a clear separation between the component behaviour and system-wide coordination policies. For coordination scenarios that require a global state information (see for example our use case below), dedicated monitor components can be added in straightforward manner. This allows to centralise all the information related to the coordination in one single location, instead of distributing it across the components. This considerably simplifies the system maintenance and improves reusability of components. Indeed, components do not carry coordination logic relying on the characteristics of any specific execution environment.

An observable trend in software engineering is that design becomes more and more declarative. Developers provide specifications of what must be achieved, rather than how this must be achieved. These specifications are then interpreted by the corresponding engines, which generate — often on the fly — the corresponding software entities. Thus, it is not always possible to instrument or even access the actual source code. Furthermore, it is usually not desirable to modify such code, since this can lead to a considerable increase of the maintenance costs.

We have taken a non-invasive approach relying, for the interaction with the controlled components, on existing API. With our approach, designers write a separate annotated Java class that we call "BIP specification". BIP specification is an abstract model of the component that is aware of its functional states. It defines the functions associated to the corresponding transitions. Transitions can be of three types: enforceable, spontaneous and internal. Enforceable transitions are used for coordination through the BIP Engine; spontaneous transitions are used to take into account the changes in the controlled component; finally, internal transitions can be used to make the BIP Specification more concise — when enabled, they are executed immediately. In order to respect the state stability assumption mentioned above, we require that transitions of only one type be enabled at any given execution step. BIP Specification developers are responsible for enforcing the validity of this requirement. However, our tools log an exception when a violation is detected at runtime.

Annotations in a BIP Specification are processed by the BIP Executor, which we have developed as part of our library, to construct a corresponding Behaviour object representing the FSM. The Behaviour object is used to coordinate the actions of the component with the other components of the system through the BIP Engine. Additional components can be provided in a similar manner to monitor and influence the system behaviour, in particular to impose safety properties. For specifying larger components, where annotations become impractical, we have defined a Behaviour API, which allows developers to construct a Behaviour object in a programmatical manner.

BIP coordination extension for OSGi has been implemented and tested in Connectivity Factory™, a flagship product of Crossing-Tech S.A. The main use-case consists in managing the memory usage by a set of Camel routes. A Camel

route connects a number of data sources to transfer data among them. The data can be fairly large and may require additional processing. Hence, Camel routes share and compete for memory. Without additional coordination, simultaneous execution of several Camel routes can lead to `OutOfMemory` exceptions, even when each route has been tested and sized appropriately on its own. We have designed an annotated BIP Specification for Camel routes, using the `suspend` and `resume` functions provided by the Camel Engine API. We also use notification policies provided by Camel to observe the spontaneous modifications of the route states. By introducing an additional Monitor component, we limit the number of Camel routes running simultaneously to ensure that the available memory is sufficient for the safe functioning of the entire system.

Introduction of OSGi was a tremendous improvement for the design of modular Java-based systems. OSGi greatly simplifies the work of a software developer and its benefit has been shown by the ever growing community of users. It takes into account such aspects as class loading, class visibility, bundle lifecycle and service dynamicity. However, the lifecycle layer makes a very strong simplification by limiting all the information about an active bundle to a single state. In Crossing-Tech's experience with OSGi, this happens to be very restrictive. In our practice, common coordination issues are very difficult to address with the mechanisms provided by the OSGi lifecycle layer. We had to resort to ad-hoc solutions to ensure that resources such as memory within the JVM running an OSGi container are not being exhausted. The specification of what are the allowed global states and global state transitions is an integral part of the specification of a modular system. Using BIP, we have shown how these aspects can be taken into account in a non-invasive manner and without any impact on the technology stack within an OSGi container following the best practices of OSGi. Although differentiating multiple components within a bundle could go beyond the desired scope of the OSGi specification, one has to notice that this finer granularity is already quite common, since OSGi best practices prefer packages to bundles as means to express dependency relationship. Thus, we consider that our work, recognizing the fact that bundles may have multiple components with multiple functional states, will help to improve the OSGi standard.

This work was co-authored by:

- Simon Bliudze, Anastasia Mavridou and Alina Zolotukhina (EPFL RiSD)
- Radoslaw Szymanek (Crossing-Tech S.A.)

This description, a technical report and the presentation slides will be available from http://risd.epfl.ch/osgi.

The presentation slides are available here - osgi-bip-eclipseco.pdf

Internal presentation at CT

This is a presentation done at the early stages of the project.

Endnotes:

1. More precisely, complete C/C++ code is generated, which is then compiled using a third party compiler for the desired target platform. [back]
2. We omit the formal definition for the sake of clarity. [back]